Systematic Quality Trade-off Support in the Software Product-Line Configuration Process

Laurens Sion,
Dimitri Van Landuyt, Wouter Joosen
iMinds-DistriNet, KU Leuven
firstname.lastname@cs.kuleuven.be

Gjalt de Jong
Melexis N.V.
gdj@melexis.com

ABSTRACT

Software product line engineering is a compelling methodology that accomplishes systematic reuse in families of systems by relying on two key principles: (i) the decomposition of complex systems into composable and reusable building blocks (often logical units called features), and (ii) on-demand construction of products and product variants by composing these building blocks.

However, unless the stakeholder responsible for product configuration has detailed knowledge of the technical ins and outs of the software product line (e.g., the architectural impact of a specific feature, or potential feature interactions), he is in many cases flying in the dark. Although many initial approaches and techniques have been proposed that take into account quality considerations and involve trade-off decisions during product configuration, no systematic support exists.

In this paper, we present a reference architecture for product configuration tooling, providing support for (i) up-front generation of variants, and (ii) quality analysis of these variants. This allows pro-actively assessing and predicting architectural quality properties for each product variant and in turn, product configuration tools can take into account architectural considerations. In addition, we provide an in-depth discussion of techniques and tactics for dealing with the problem of variant explosion, and as such to maintain practical feasibility of such approaches.

We validated and implemented our reference architecture in the context of a real-world industrial application, a product-line for the firmware of an automotive sensor. Our prototype, based on FeatureIDE, is open for extension and readily available.

CCS Concepts

•**Software and its engineering** → *Software configuration management and version control systems; Software design tradeoffs;*

SPLC '16, September 16 - 23, 2016, Beijing, China

ISBN 978-1-4503-4050-2/16/09...$15.00

DOI: http://dx.doi.org/10.1145/2934466.2934481

1. INTRODUCTION

In its essence, Software Product Line Engineering (SPLE) is a systematic divide-and-conquer approach for dealing with software complexity: a family of systems is decomposed into composable building blocks (typically logical units called *features*), and individual products of such product lines are constructed from these building blocks, allowing for cross-product feature reuse, modular evolution and maintenance, and cost reductions through economy-of-scale benefits. Such processes are commonly guided by activities of requirement analysis, requirements selection, and product configuration in terms of pre-defined features [21]. However, not all variability resides in the requirements space, and in many cases configuration decisions have architectural impact. This distinction is often called *internal* versus *external variability* [21], whereas internal variability is also called *realization-driven variability* [19].

In many cases, configuration decisions in software product lines have large impact on certain qualities of the ensuing product variant. However, traditional variability modeling techniques provide little support for quality trade-offs: the limited information available thus makes it hard to grasp the impact or consequences of a configuration decision. In their study on SPLE adoption in the industry, Deelstra et al. explicitly highlight this problem as a hurdle to practical adoption [6]. Over the years, there has been substantial research focus on addressing these issues, giving rise to a wide range of techniques that bring software quality considerations into the equation [9, 10, 17, 23, 25, 24, 32, 33]. Most of these approaches however involve *generalization* of quality-related information up to the level of features, as such extending and enriching the variability model itself, e.g., by enriching the feature model with quality information. The major disadvantages of such an approach are: (i) as the impact of specific features on the product quality might be highly dependent on external factors and other decisions, generalization might be inappropriate, (ii) it tightly couples the variability model with information about product variants and as such hinders the independent evolution of features and the core assets that they represent.

In this paper, we present an alternative, data-driven approach that involves storing product variant properties at large scale in a variant inventory. The main idea is (i) to apply *generative* techniques (in a brute force, or targeted fashion), i.e. rendering up-front the outcome of the product derivation step at a sufficiently representative level of accuracy (e.g., architectural models), then, (ii) to run a number of automated variant analysis activities, (iii) the outcomes of

which in turn *complement* (instead of extend) the variability model with per-variant quality information. Our approach measures variant properties in order to make statements about certain software qualities.

This approach contrasts with those presented above, because it focuses on the generated end product instead of an enriched feature model, as such enabling a property-driven selection process. Additionally, the focus on properties of the end products enables the inclusion of external and non-generalizable effects in the decision process. We present our reference architecture for generic tooling to support such fundamental architectural decisions, which includes an in-depth discussion of tactics and techniques that can be employed to avoid issues related to the combinatorial explosion of variants (*variant explosion*) that may drastically impact practical feasibility and scalability of this approach, e.g. [14, 20, 9]. We systematically classify these tactics in four distinct categories: scoping, abstraction, sampling, and optimization; and we discuss their impact on the approach.

We have motivated and validated this work in the context of an industrial case, the firmware design of a Hall Effect sensor that is integrated in a number of automotive applications [31, 5]. This involves a number of essential trade-off decisions for which unexperienced engineers (or engineers less aware of the ins and outs of the individual features) may require extra support.

The approach is implemented and validated in our variant selection tool, which is an extension to FeatureIDE [30]. The tool is designed to be generic and extensible in the types of analysis activities that are supported and shows how quality trade-offs are supported in practice.

The remainder of this paper is structured as follows: Section 2 motivates this work and defines the main problem statement. Section 3 presents our overall approach, and Section 4 illustrates its application in the safety design of the Hall Effect sensor mentioned above. Section 5 discusses our prototype implementation and presents our evaluation. Section 6 discusses related work, after which Section 7 concludes the paper.

2. MOTIVATION

The motivating case for this paper focuses on the design of an integrated Hall Effect sensor used in automotive systems [5, 31]. A Hall Effect sensor is a transducer that varies its output voltage in response to a magnetic field, allowing sensors to calculate angles in two- or three-dimensional spaces. As illustrated in Figure 1, this product is currently integrated in a wide range of automotive applications with varying safety requirements, ranging from windshield wipers, to brake or gas pedals. The safety-criticality of the sensor depends highly on the nature of the automotive application into which it is integrated.

In earlier work [31], we have discussed the different forms of variability at play in the creation of a software product line for the firmware of this sensor, among which: (i) variability in terms of the functional requirements of the automotive application in which the sensor is integrated and (ii) variability in the safety design. In the context of a research project in collaboration with industry [1], we have systematically modeled these forms of variability as feature models.

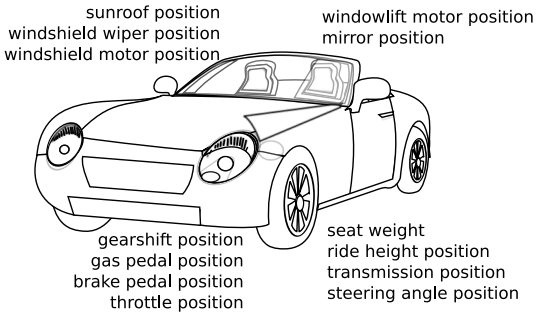


Figure 1: Overview of the different automotive applications involving the MLX90365 Hall Effect sensor.

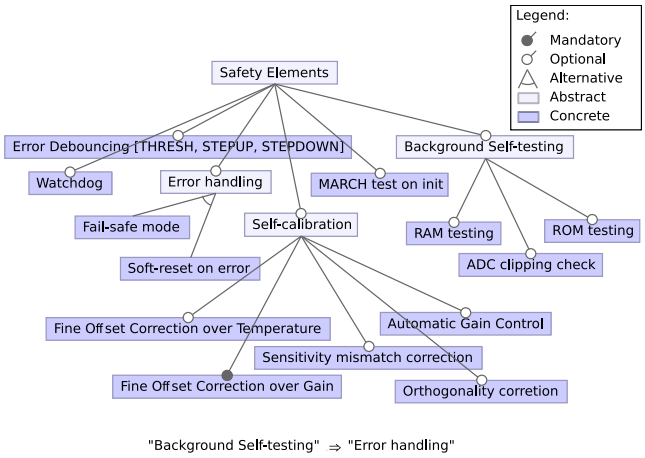


Figure 2: Internal feature model of safety elements offered by the MLX90365 Hall Effect sensor (simplified)

Functional variability

This involves customizing the functionality of the sensor to the automotive application at hand, for example by configuring the plane in which to calculate the angle (e.g., 2- or 3-dimensional), the activation of specific transformations, ...

Variability in the safety design

The *Functional Safety for Road Vehicles* standard (ISO 26262 [29]) is the applicable safety standard defining a risk classification scheme called the Automotive Safety Integrity Level (ASIL), in which the assessment of hazard factors is based on the relative impact of hazardous effects related to a system (Severity), and the relative likelihood of the hazard manifesting those effects (Exposure and Controllability). ASIL defines five Risk Levels, ranging from ASIL-D, the strictest level, to ASIL-QM (Quality Management, i.e. there is no need of safety governance by ISO 26262). The current firmware implementations provide a number of safety-related mechanisms such as a watchdog, error debouncing, ROM checksumming, etc. A simplified version of the feature model that represents a number of these optional safety solutions, supported by the firmware, is presented in Figure 2.

Problem statement

Offering configuration interfaces to the customer or integration engineer that are based exclusively on variability models (such as the feature model in Figure 2) is insufficient: the stakeholder responsible for configuration is offered no feedback about the effects of his decisions, about the combinations of features that work best, about the alternatives that might be better in terms of specific qualities, or about the trade-offs involved.

This is especially troublesome in combination with the problem of *variant explosion*: analysis of the feature models we have designed of the Hall Effect sensor (which cover only around 29% of all configuration options in the real-world product) shows that the total amount of valid product variants is 3 498 048, which clearly poses a threat on the practical feasibility of an SPLE-based solution that is built around these feature models.

3. APPROACH

Figure 3 presents an overview of the activities and artifacts involved in our approach, more specifically depicting three Application Engineering activities: (i) Variant Generation, which involves the systematic generation of variants, (ii) Variant Analysis, which involves the deduction of variant properties of interest, and (iii) Variant Selection, which involves using the variant properties to guide the configuration process. The remainder of this section discusses these key activities in more detail, respectively in Sections 3.1, 3.2, and 3.3. Throughout this section, we refer to Figure 4, which presents a tactics tree, indexing different tactics to counter the problem of variant explosion, and Figure 5 which depicts the artifacts involved in these Application Engineering activities.

3.1 Variant Generation

This activity involves the up-front generation of product variants. These derivations are based on the feature model and the different potential configurations it supports, illustrated on the left-hand side of Figure 5. The main outcome of this activity is a set of product variants, each associated with its configuration. They are stored in the `VariantInventory` and serve as input for subsequent analysis and selection.

Listing 1 shows the pseudo-code of the generation step. The generator iterates over all valid configurations, and composes the product based on the selected features.

Listing 1: Pseudo-code for the Variant Generation activity.

```
1  FeatureModel fm = loadFM('model.xml');
2  VariantInventory vi = new VariantInventory(fm);
3  Variant var = new Product();
4  for (config:fm.getValidConfigurations()){
5      for (Feature feature : config){
6          var.compose(
7              getReusableAsset(feature)
8          );
9      }
10     vi.store(var);
11 }
```

Feasibility tactics: The following tactics can be applied to ensure practical feasibility of this activity: (i) line 1: controlling the size or complexity of the feature model will have a large impact on the enumeration of potential product variants (**scoping** tactic, applied to the feature model), (ii) line 4: the algorithm to enumerate all the valid configurations (`getValidConfigurations()`) can be replaced with techniques of sampling to obtain a representative subset of configurations (**sampling** tactic, applied to the products), (iii) lines 6–7: the actual generation of variants can be performed at a higher level of abstraction (e.g., model-driven, as in Section 4) (**abstraction** tactic, applied to the product), (iv) lines 4–10: the generation of variants can be optimized by, for example, hierarchically traversing the feature model, and reusing information of generated variants for features they have in common (**optimization** tactic, applied to the generator).

3.2 Variant Analysis

The Variant Analysis activity involves further analysis and investigation of the variants created during the Variant Generation, more specifically leading to the instantiation of a list of `VariantProperties` per variant. As is depicted at the center of Figure 5, a `VariantProperty` allows to generically express information about a single `Variant`. It has a selected set of abstract types (i.e., `Boolean-`, `Enum-`, `Number-`, and `StringBasedVariantProperty`) which are specialized as desired, depending on the type of the property that needs to be represented.

Our design is generic and extensible: the `VariantInventoryAnalyser` refers to a list of `VariantAnalysisActivities`. The `VariantAnalysisActivity` is an application of the strategy pattern. It defines an abstract method to retrieve or calculate specific `VariantProperties` for a concrete `Variant`, as shown in Figure 5. Since some analysis activities involve directly comparing an individual variant to its siblings, a reference to the `VariantInventory` is passed along. Introducing new analysis activities simply involves specializing `VariantAnalysisActivity`, and new variant properties can easily be supported by specializing the `VariantProperty` classes.

Listing 2: Pseudo-code for the Variant Analysis activity.

```
1  VariantInventory vi; // as initialized before
2  for (Variant variant:vi.products){
3      for (AnalysisActivity AAct :
4          getAnalysisActivities()){
5          vi.put(
6              variant,
7              AAct.getVariantProps(
8                  variant,
9                  vi));
10     }
11 }
```

As shown in Listing 2 (lines 3–4), the `VariantInventoryAnalyser` iterates over the `VariantAnalysisActivities`, calculates the properties for the individual variants in the inventory (lines 7–9), and adds these properties to the `VariantInventory` (line 5).

Feasibility tactics: The following tactics (cf. Figure 4) can be applied in this activity: (i) lines 3–4 could be parallized, by dividing the set of variants over multiple machines (**optimization** tactic, applying parallelism), (ii) lines 2 and 3 could be swapped, allowing the usage of more advanced analysis techniques that involve caching or reusing information from previous analysis runs (**optimization** tactic, applying reuse), (iii) similarly, lines 2 and 3 could be swapped, allowing the creation of dedicated worker nodes, specialized in a specific analysis activity (**optimization** tactic, applying

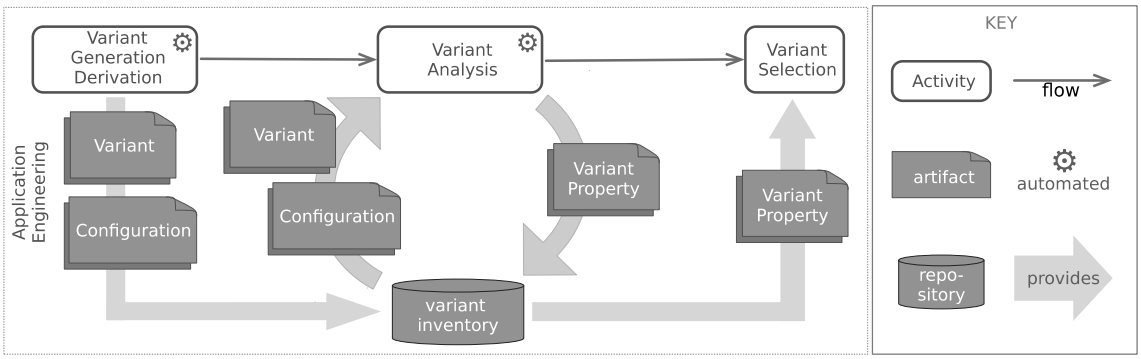


Figure 3: Overview of the presented approach.

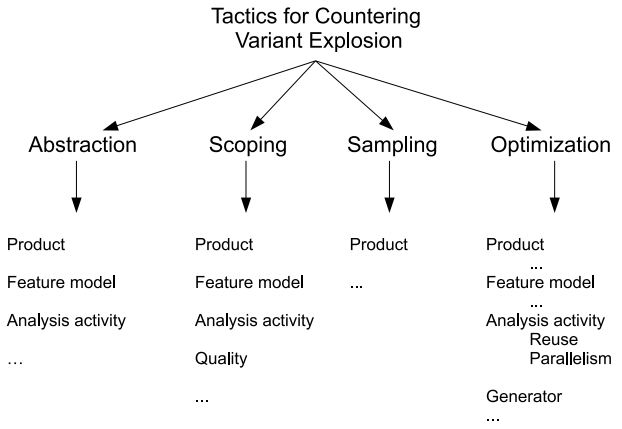


Figure 4: Tactics for countering variant explosion

parallelism), (iv) line 7: the nature of the analysis activities themselves can be of differing complexity, ranging from (semi-)automated metrics to manual analysis techniques, requiring input from a human expert (**abstraction** tactic, applied to the analysis activity), (v) line 7: the analysis activites can also be modified to obtain a more narrow set of variant properties (**scoping** tactic, applied to the analysis activities).

3.3 Variant Selection

After enriching the `Variants` in the `VariantInventory` with properties, the Variant Selection activity employs this information to guide the engineer in the selection process. For this, we envision tool support for generating a view on and on-demand querying of the `VariantInventory`. Because of the generic and extensible way in which variant properties are defined, the view does not rely on any a-priori knowledge of the subtypes below `Boolean-`, `Enum-`, `Number-`, and `StringBasedVariantProperties`.

As shown in the right-hand side of Figure 5, the view is dynamically constructed, based on filters for each property (sub)type, grouped according to the four types mentioned above. In addition to these filters, there is also a `Feature-Filter` which is used to check which variants match the (incomplete) configuration, the user has opened. Filters are used to define specific constraints according to which the view is constructed.

Variant property-related trade-off scenarios

After loading the variant inventory, the view splits the set of candidate variants into two subsets: the set containing the variants that match the constraints set by the active filters (and therefore are included), and the set containing the variants that don't match (and therefore are excluded). Constructing views in this manner allows for several types of interesting queries to explore the list of potential product variants.

1) **Impact of feature selection:** When the engineer selects or excludes a certain feature from the configuration, the sets of complying and non-complying variants change; this allows the engineer to consider the impact of a specific configuration decision, by studying the changes in the variant properties in both sets.

2) **Impact of setting variant property constraints:** Additionally, setting filters on specific variant properties allows the engineer to observe how these filter constraints impact the availability of features (i.e., the opposite direction). In this case, the desired properties of the end product are used to reduce the set of potential variants. This makes it possible to calculate which configuration decisions those variants have in common; and as such allowing the engineer to learn about how certain quality constraints can mandate or prohibit certain features.

3) **Finding correlations between variant properties:** In the final example usage scenario, the engineer can explore the correlations between different variant properties, and by extension, the software qualities they represent. Contrary to the previous scenario where we look at the impact on the sets of included and excluded variants, we focus specifically on how one filter impacts other variant properties and qualities. This allows the engineer to gain more insight in the product and the qualities.

Feasibility tactics: The following tactics (cf. Figure 4) can be applied in this activity: (i) representing the variant property information at a higher level of abstraction (**abstraction** tactic, applied to the variant properties), (ii) limiting the set of variant properties (**scoping** tactic, applied to the variant properties), (iii) using techniques such as information gain to guide the selection process based on variant properties that best divide the set of variants (**optimization** tactic, applied to the variant properties and selection process).

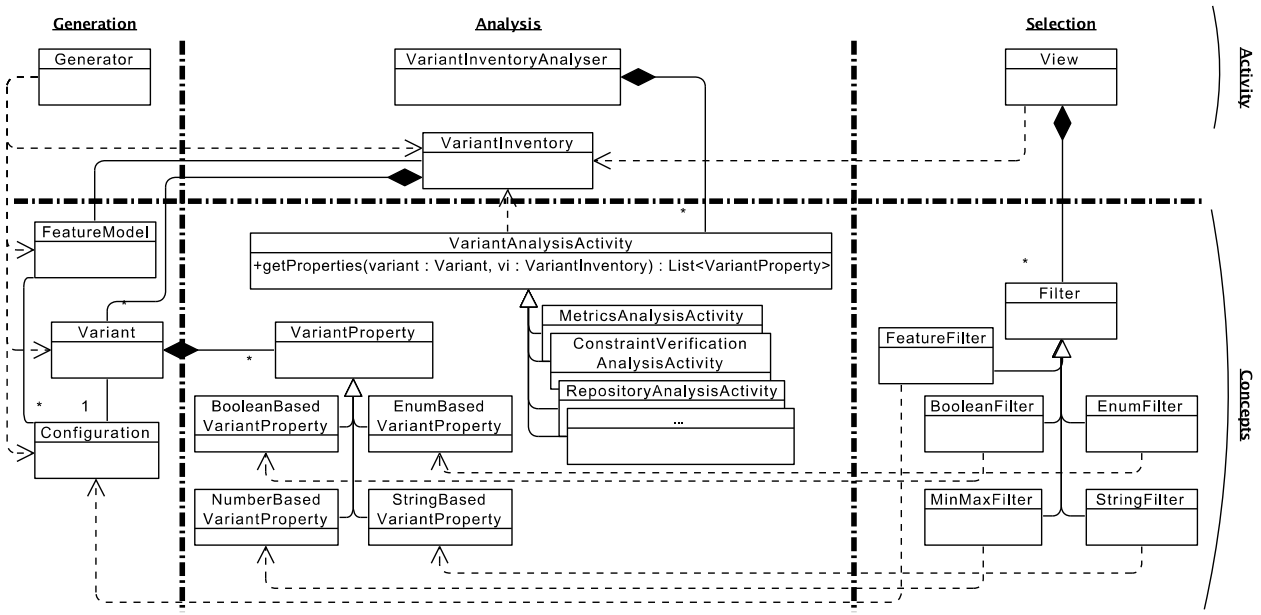


Figure 5: Approach Activities and Meta-Model Concepts.

4. APPLICATION TO THE AUTOMOTIVE HALL EFFECT SENSOR

We validated our approach in the context of a real-world industrial application, a product line for the firmware of the automotive Hall Effect sensor introduced in Section 2. The remainder of this section discusses in further detail the key design decisions (Section 4.1), the Variant Generation activity (Section 4.2), the Variant Analysis activity (Section 4.3) and finally the Variant Selection activity (Section 4.4). An in-depth demonstration of our prototype (including source code) can be found via [27].

4.1 Key Design Decisions

To maintain practical feasibility, we have made the following important design decisions, as such applying some of the tactics of Figure 4 in the context of this industrial case:

1) **Feature model scoping:** In order to obtain a manageable number of generated variants, we (a) focused purely on safety design variability, addressing external variability beforehand in a customer-driven configuration step [31] (**scoping** tactic, applied to the feature model), (b) spent specific attention during the design of the internal feature model that it does not span an overly large variability space (**scoping** tactic, applied to the feature model), and (c) introduced additional constraints as to control the variant explosion problem, e.g., by not considering *all* potential variants when only an ASIL-C design is required (**scoping** tactic, applied to the feature model).

2) **Product-level abstraction:** Instead of directly generating firmware code, we generate architectural models that represent different architectural variants (**abstraction** tactic, applied to the product). This abstraction proves useful in the derivation activity, yet still allows for meaningful model-based variant analysis activities.

3) **Analysis-level abstraction and scoping:** As a consequence of the previous decision, we applied model-driven, quality-predictive techniques, instead of performing quality assessments on the final products (**abstraction** tactic, applied to the analysis activities). Additionally, we limit the analysis effort by focusing, in the variant analysis, on a clearly defined list of software qualities – mainly safety and related qualities, such as schedulability, accuracy, and robustness (**scoping** tactic, applied to the analysis activities).

4.2 Variant Generation

The Variant Generation activity in the Hall Effect sensor case involves using different model-based composition and pattern instantiation techniques such as: (i) the inclusion or exclusion of specific model elements from a base model, and (ii) the application of architectural patterns:

Base model: A Hall Effect sensor is essentially a signal processing application; it is therefore not surprising that its core design is based strongly on architectural patterns such as pipe-and-filter, and filter chaining. Filters are chained in a linked list and therefore provide the main abstraction, realizing modularity, pluggability and extensibility. We rely on these properties to generate specific variants, more specifically using KCVL [7] which is a set of derivation tools created around an implementation of the Common Variability Language (CVL).

Architectural patterns: In addition, we employ architectural pattern instantiation technology [7] to apply specific architectural safety patterns (e.g., the watchdog pattern) in an automated fashion.

4.3 Variant Analysis

In this activity, we have integrated a number of model-based analysis techniques to quantify and predict additional desired qualities (other than safety) such as schedulability,

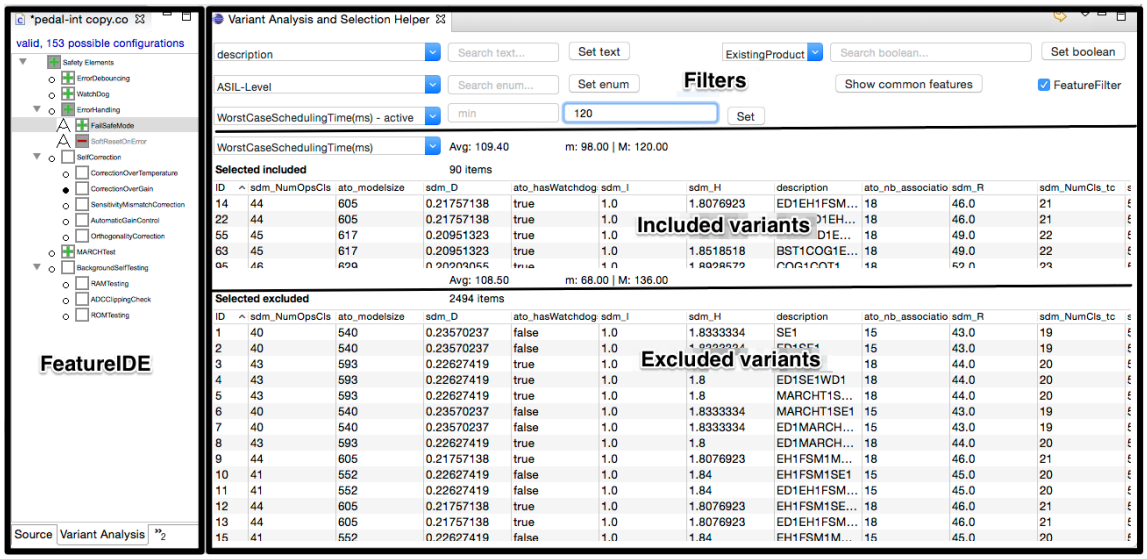


Figure 6: Screenshot of the trade-off support tool prototype.

resource utilization, and firmware size. Below, we list the model-based Variant Analysis activities we have practically integrated. The resulting `VariantProperties` are all added to the `VariantInventory`.

1) **Constraint verification:** In this analysis activity, the generated alternative architectures are verified w.r.t. specific constraints. These constraints represent invariant design conditions that should be satisfied in the resulting, composed design. We make the distinction between domain constraints (to restrict the overall design), and safety constraints (to introduce specific restrictions for safety). We employ model-based testing techniques to verify syntactic and semantic correctness of the different alternatives.

2) **Model metrics:** These analysis activities involve gathering additional design-level information, by using (i) standalone model metrics for model size, complexity (coupling/cohesion) and (ii) comparative model metrics (model matching and differencing). The use of these metrics is motivated from a number of guiding principles. For example, if one candidate model is entirely subsumed by another model, the latter is probably a less suitable candidate since the former accomplishes the same in terms of functionality and safety, yet is smaller.

3) **Resource utilization prediction and analysis:** Since a HES is a resource-constrained chip (CPU, RAM, storage), not including unnecessary features is favored; or, in other words, the size of the generated firmware is a key factor. To take this into account, we also include model-based predictive analysis techniques in the Variant Analysis activity, based on meta-data properties detailing the projected size or weight of specific model elements (after compilation, in terms of firmware size),

4) **Schedulability analysis:** To make sure that the scheduled task can in fact be executed within the time frame of a single angle acquisition window (which is fixed), dedicated schedulability analysis is conducted. This analysis activity involves transforming the models (together with meta-data about the worst-case timings of specific method executions) into an AADL [8] model which is then used for schedulability analysis [28].

5) **Predicted Firmware size:** This analysis activity involves estimating the size of the compiled firmware from the generated models.

4.4 Variant Selection

To assist the the engineer, a stakeholder in the role of product architect or product manager in the variant selection process, we have created tool support, which is an extension to FeatureIDE [30]. Figure 6 presents a screenshot of this prototype. The tool allows the engineer to select a variant, not purely at the basis of the feature model (shown in the left pane), but by setting specific filters (top pane on the right) and consulting the sets of included (middle pane at the right) and excluded (bottom pane on the right) variants:

1) **FeatureIDE configuration editor:** This pane, situated on the left in Figure 6, is a modified version of the FeatureIDE configuration editor. It allows the engineer to explore the different features by selecting or excluding them, and to consider the impact on the resulting variants.

2) **Filters**: The top-right pane in Figure 6 provides controls to the filters for the different types of variant properties (i.e., text, enum, numeric, boolean). It allows the engineer to activate and stack multiple filters, to tweak different parameters, for example to set certain intervals. A special type of filter is the feature filter controlled by the FeatureIDE configuration editor on the left.

3) **Included variants**: The middle pane on the right shows the set of all currently-included variants, i.e. it lists all variants that match all the activated filters. In addition, this part of the tool also presents the engineer with some common statistics (e.g., average, min, max) for a selected numerical metric. This allows the engineer to consider the impact of selecting a feature or setting a certain filter, on some other desired metric.

4) **Excluded variants**: Similarly, the bottom pane on the right shows the set of all excluded variants, i.e. those variants in the variant inventory currently excluded by the activated filters. It also presents some statistics and thus allows the engineer to compare these metrics and considering the impact of a selection or filter on that metric.

Variant property-related trade-off scenarios

The controls described above enable a number of trade-off decision and exploration scenarios, i.e. those discussed earlier in Section 3.3:

1) **Impact of feature selection:** Since the feature selection in the FeatureIDE configuration editor (on the left) works as a filter on the variants, any modifications can immediately be perceived in the metrics tables. This allows the engineer to consider the impact of enabling or disabling certain features in the configuration, by evaluating how a feature selection changes the values of the relevant variant properties.

2) **Impact of setting variant property constraints:** By instantiating different filters on the view, the engineer can limit the set of variants under consideration to those that actuality meet the required quality constraints. Additionally, this information can be translated back to the feature model by letting the tool calculate the features all the relevant variants have in common (via the "*Show common features*" button on the top right). This allows the engineer to start from a common set of features that are required because of the quality constraints, and to focus on the open features to further optimize the final selection.

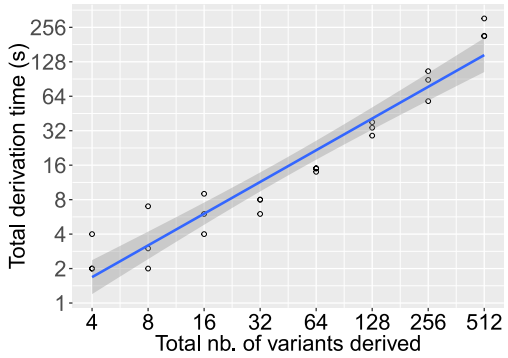
3) **Finding correlations between variant properties:** In the final example usage scenario, the engineer can explore the correlations between certain variant properties and qualities. By setting specific variant property filters, the engineer can explore how these restrictions affect the other variant properties and qualities. For example: *"Does restricting the set to ASIL-C have an impact on the firmware size?"*, *"Do smaller firmwares have shorter WorstCaseScheduling timings?"*. This allows the engineer to gain more insight into how different qualities impact each other and provides him with direct feedback on which filters might prove most useful.

5. EVALUATION

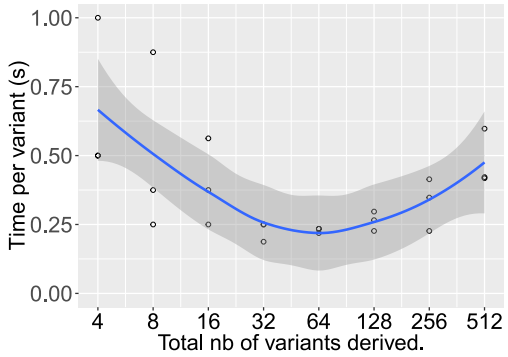
Section 4 illustrated the practical feasibility of our approach up to the level of a working prototype which is applied to an industrial application, a product line for the firmware of an automotive sensor. However, since our approach involves generation and analysis of products at large scale, the practical feasibility in terms of performance and scalability (in the number of variants) remains a concern. Section 5.1 therefore presents the results of our performance measurements, as such indicating the feasibility of our approach. Section 5.2 subsequently elaborates on the necessity of the feasibility tactics discussed in Section 4.1.

5.1 Performance and scalability

Table 1 presents the performance measurements that were obtained in the context of our prototype and the industry case, more specifically by running the prototype on a machine with a 2.6 GHz Intel Core i5 with 8 GB, 1600MHz DDR3, and a 256GB SSD drive. The presented results are averaged over at least three distinct runs. As shown, the largest performance cost is associated to the Variant Generation activity: the average derivation time (per variant) ranges between 229ms and 667ms, while the Variant Analysis activity requires only around 23ms per variant. This includes iterating over the variants and performing five distinct analysis activities, loading the UML-models from file, and saving all the variant properties in the variant inventory XML-file.



(a) Total derivation time



(b) Derivation time per variant

Figure 7: Performance of the Derivation step

Figure 7 zooms in further on the required derivation time. Figure 7a shows how the total derivation time scales linearly with an increasing number of variants, while Figure 7b shows the derivation time per individual variant.

The performance evaluation of the generation activity highlights other noteworthy aspects: (i) when generating smaller amounts of variants, the average time required to generate a single variant is relatively high. This can be explained by the overhead in bootstrapping the derivation process, and (ii) from a certain scale (in our case, starting from the generation of 128 variants), the average time to generate a single variant increases again: this can be explained by the increased memory usage and CPU contention on a single machine.

These results show that the Generation and Analysis activities of our approach do come at a certain performance cost. It must however be noted that (i) these activities only have to be performed once for a certain product-line and the construction of the variant inventory can be done up front as a batch operation, (ii) computational power and storage resources are becoming increasingly cheap, as opposed to for example human analysis effort, and (iii) there are many unexplored opportunities for optimizing the derivation process, e.g., reusing (partial) derivation results for sub-features or parallelizing the execution of the Generation and Analysis activities.

5.2 Practical feasibility

Section 4.1 discussed the application of a number of tactics (Figure 4) that target specific sources of variant explosion

Type	Nb. of variants	Total time (s)	Time per variant (ms)
Generation /	4	2.667	666.667
Derivation	8	4.000	500.000
	16	6.333	395.833
	32	7.333	229.167
	64	14.667	229.167
	128	33.667	263.021
	256	84.333	329.427
	512	245.333	479.167
Analysis	2584	58.840	22.770

Table 1: The results of our performance measurements of the Variant Generation and Analysis activities.

problems and as such were required to ensure the practical feasibility of our approach in the context of the design of the Hall Effect sensor: (i) **scoping** applied to the feature model**,** (ii) **abstraction** applied to the product, and (iii) **abstraction** and **scoping** applied to the analysis activities. We systematically assess the value of these tactics below by considering the situation in which these tactics would not be applied.

1) **Feature model scoping:** Table 2 compares the number of potential variants that are valid in the context of the safety feature model (second column) with the number of potential variants in the feature model that combines functional variability and safety variability (third column). These numbers clearly illustrate the value of the feature model scoping tactic: by only focusing on variability in the safety design, the total number of variants is drastically reduced from 3 498 048 variants to only 2 584 variants (a reduction of around 99.93%). As such, the required effort in the variant generation, analysis and section activities is vastly reduced. A second decision, forcing the customer to define the desired ASIL-level up-front, reduces the total number of variants even further, as illustrated in the different rows of Table 2 which present the amounts of possible variants for each safety level in increasing level of strictness.

2) **Product-level abstraction:** We have applied the tactic of making abstraction to the product, more specifically by generating architecture models instead of code, or even compiled and linked firmware. Based on numbers obtained from the production environment, compiling and linking the firmware takes in the order of magnitude of 2 minutes per variant. Software simulation (regression testing) takes approximately 2 hours. Applying these numbers to the amounts of variants that are considered (cf. Table 2) clearly shows that a code-driven product variant generation would drastically impact the practical feasibility of the proposed generative approach: not taking into account regression testing, generation would take around 86 hours, while it would take 5 250 hours to generate code-level variants and perform the regression tests.

3) **Analysis-level abstraction and scoping:** The performance measurements presented earlier in Section 5.1 indicate that we managed to keep the cost of variant analysis within bounds. This can be attributed to (i) the tactic of making **abstraction** of the analysis activities, more specifically by focusing on integrating variant analysis activities that are predictive and lightweight and (ii) the tactic of

Safety Level	Variability in	
	Safety design	Functionality and safety design
ASIL-QM	**2 584**	**3 498 048**
ASIL-A	544	2 009 088
ASIL-B	128	755 712
ASIL-C	32	165 888
ASIL-D	8	18 432

Table 2: Illustration of *variant explosion*: overview of the amounts of possible configurations for each safety level.

scoping the analysis activities, i.e. by focusing on a select set of qualities that are related to safety (e.g. schedulability, etc.).

To illustrate with some statistics from the industry case, executing a full-blown safety assessment and verification takes several weeks for a single variant. Doing this systematically and sequentially for all 2 584 variants would thus take over fourteen years. Evidently, the analysis activities implemented in our prototype are not a replacement of such manual safety assessment and verification exercises, but are focused at the automated gathering of additional variant properties that will provide the engineer with additional information to support variant property- and quality-aware variant configuration and selection.

6. RELATED WORK

This section discusses related work. First, we outline the body of existing research on quality predictive techniques in software product lines. Subsequently, we discuss related approaches that also deal with the problem of variant explosion, such as search-based testing. Finally, we discuss related work from the research area on software architecture and trade-off analysis support.

Quality prediction in software product lines

Zhang et al. [33] were among the first to stress the importance of quality considerations in the context of systematic product line approaches. Their approach involves (i) obtaining inputs from domain or application experts and (ii) using this knowledge to construct a Bayesian Belief Network that can be consulted to assess or predict the impact of a design decision (a variant) on software quality. The main difference to our approach is that the quality prediction and analysis is done on-demand (by performing quantitative analysis over the BBN), whereas our approach involves executing such analysis activities up-front, and consolidates the results in the central variant inventory.

Siegmund et al. [25, 23, 24] adopted a highly similar approach to ours: deriving the impact of specific features on non-functional properties and software quality in general, using techniques of variant analysis and quality prediction. Also similar is the *feedback approach* proposed by Sincero et al. [26], which involves generating a limited set of representative products (the testing set), and performing benchmark tests to assess the quality implications of specific configuration decisions, as such yielding per-feature quality knowledge that is persisted in a central knowledge database (the NFP DB). The main difference to our work is that these approaches perform *generalization* of the gathered results, by enriching

the feature model itself, or by introducing feature-level metadata about the impact of these features on specific software qualities. Our approach in turn does not perform generalization as such: our variant inventory collects metrics for all the possible variants (or a representative subset thereof), and our tool uses the raw data in the variant inventory as a knowledge base to guide side-by-side comparison and selection of individual variants.

Lillack et al. [17] explore the value of including contextual parameters in the analysis activities – i.e., specific environmental factors that should be taken into account when assessing the impact of specific features on non-functional properties of a system.

Variant explosion

The problem of combinatorial explosion as a consequence of systematic variability modeling has been recognized and received considerable attention in recent years [15].

Many techniques are based on product similarity assessment and similarity-based prioritization, specifically to reduce the amount of product variants to investigate. Hajjaji et al. [4, 3] combine sampling techniques with similarity-based prioritization, whereas Perrouin et al. [20] and Henard et al. [14] apply similarity-based prioritizing in the context of pair-wise testing.

Ghezzi et al. [10, 9] investigate the potential of employing parameterized model checking techniques instead of model checking on a per-variant basis, which seems to be a promising tactic to flatten the scalability curve.

Similarly, Zhang et al. [32] employ an analytic hierarchical process to systematically incorporate domain expert knowledge on software quality into software product lines, without having to iterate over all possible product variants.

In contrast, our approach does in fact involve generating many (if not all) potential product variants, but focuses on the one hand on maintaining the practical feasibility by means of a diverse array of tactics, and on the other hand on large-scale variant inventories to support our tool.

Trade-off analysis and decision support

In the broader software engineering research domain, many approaches exist to support making key trade-off decisions, including techniques and tools that support multi-objective optimization [11, 12]. The research domain of search-based software engineering and search-based testing [2, 22, 13] involves the search of suitable meta-heuristic search techniques such as evolutionary algorithms [16, 18] to more efficiently explore the solution space. Instead of a brute-force iteration of the search space, integration and introducing support for such techniques into our reference architecture represents a specific dimension of architecture variability that we aim to cover in future work.

7. CONCLUSION

Especially in situations where specific configuration decisions have significant architecture-level impact on software quality, making informed configuration decisions is of utmost importance. In this paper, we presented our generic approach and corresponding reference architecture for generative and quality-predictive or quality-analytic techniques, which in essence adopts a *big data* approach: instead of generalizing or consolidating analysis results into standalone decision or trade-off support structures, we store the obtained raw analysis results in the variant inventory, which is a per-variant knowledge base on top of which such decision support structures and tools can be built.

In addition, we present our variant configuration tool, an extension to FeatureIDE, which allows the architect to explore the potential trade-offs during product configuration. We have validated and evaluated these techniques in the context of an industrial application, the safety design of an automotive Hall Effect sensor, and we provide an in-depth discussion on the tactics and techniques that can be employed to ensure practical feasibility and maintain scalability of such a brute-force, generative approach.

In future work, we will apply these techniques in the context of dynamic software product lines and self-adaptive systems, in which the gathered information centralized in the variant inventory will serve as the basis for automated reconfiguration decisions, as opposed to decisions made by human architects. In addition, we will further explore techniques to address the problems of variant explosion, for example by exploring techniques to identify feature orthogonality with respect to specific qualities, i.e. by applying statistical testing techniques to identify correlations between different individual metrics. Finally, we will explore techniques that leverage the obtained per-variant quality information to apply product-line optimization, by enabling feature model refactorings such as removing unnecessary features and detecting previously-unknown feature interactions.

Acknowledgements. The presented research is partially funded by the Research Fund KU Leuven and the Flemish agency for Innovation by Science and Technology (IWT 120085). The research activities were conducted in the context of ITEA2-MERgE (Multi-Concerns Interactions System Engineering, ITEA2 11011) [1].

8. REFERENCES

[1] MERgE: Multi-Concerns Interactions System Engineering. http://www.merge-project.eu/.

[2] W. Afzal, R. Torkar, and R. Feldt. A systematic review of search-based testing for non-functional system properties. *Information and Software Technology*, 51(6):957–976, 2009.

[3] M. Al-Hajjaji. Scalable sampling and prioritization for product-line testing. In *Software Engineering & Management*, pages 295–298, 2015.

[4] M. Al-Hajjaji, T. Thüm, J. Meinicke, M. Lochau, and G. Saake. Similarity-based prioritization in software product-line testing. In *Proceedings of the 18th International Software Product Line Conference - Volume 1*, SPLC '14, pages 197–206, New York, NY, USA, 2014. ACM.

[5] G. F. Close and G. de Jong. Model-based progressive design and verification of an integrated CMOS magnetic sensor for automotive applications. In *Proceeding of the 2012 Forum on Specification and Design Languages, Vienna, Austria, September 18-20, 2012*, pages 239–245, 2012.

[6] S. Deelstra, M. Sinnema, and J. Bosch. Experiences in software product families: Problems and issues during product derivation. In *Software Product Lines*, pages 165–182. Springer, 2004.

[7] T. Degueule, O. Barais, M. Acher, J. Le Noir,

S. Madelénat, G. Gailliard, G. Burlot, O. Constant, et al. Tooling support for variability and architectural patterns in systems engineering. In *Proceedings of the 19th International Conference on Software Product Line*, pages 361–364. ACM, 2015.

[8] P. H. Feiler, D. P. Gluch, and J. J. Hudak. The architecture analysis & design language (AADL): An introduction. Technical report, DTIC Document, 2006.

[9] C. Ghezzi and A. M. Sharifloo. Verifying non-functional properties of software product lines: Towards an efficient approach using parametric model checking. In *Software Product Line Conference (SPLC), 2011 15th International*, pages 170–174. IEEE, 2011.

[10] C. Ghezzi and A. M. Sharifloo. Model-based verification of quantitative non-functional properties for software product lines. *Information and Software Technology*, 55(3):508–524, 2013.

[11] L. Grunske. Identifying good architectural design alternatives with multi-objective optimization strategies. In *Proceedings of the 28th international conference on Software engineering*, pages 849–852. ACM, 2006.

[12] L. Grunske. Early quality prediction of component-based systems - a generic framework. *J. Syst. Softw.*, 80(5):678–686, May 2007.

[13] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *Software Engineering, IEEE Transactions on*, 36(2):226–247, 2010.

[14] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. L. Traon. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test suites for large software product lines. *arXiv preprint arXiv:1211.5451*, 2012.

[15] A. Legay and E. P. de Vink. Splat 2014: First international workshop on software product line analysis tools. In *SPLC*, page 353, 2014.

[16] R. Li, R. Etemaadi, M. T. Emmerich, and M. R. Chaudron. An evolutionary multiobjective optimization approach to component-based software architecture design. In *Evolutionary Computation (CEC), 2011 IEEE Congress on*, pages 432–439. IEEE, 2011.

[17] M. Lillack, J. Mueller, and U. W. Eisenecker. Improved prediction of non-functional properties in software product lines with domain context. In *Software Engineering*, pages 185–198. Citeseer, 2013.

[18] A. Martens, H. Koziolek, S. Becker, and R. Reussner. Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, pages 105–116. ACM, 2010.

[19] R. Mietzner, A. Metzger, F. Leymann, and K. Pohl. Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications. In *Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*, pages 18–25. IEEE Computer Society, 2009.

[20] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. Le Traon. Pairwise testing for software product lines: comparison of two approaches. *Software Quality Journal*, 20(3-4):605–643, 2012.

[21] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.

[22] A. Reuys, E. Kamsties, K. Pohl, and S. Reis. Model-based system testing of software product families. In *Advanced Information Systems Engineering*, pages 519–534. Springer, 2005.

[23] N. Siegmund, M. Rosenmueller, C. Kaestner, P. G. Giarrusso, S. Apel, and S. S. Kolesnikov. Scalable prediction of non-functional properties in software product lines. In *Software Product Line Conference (SPLC), 2011 15th International*, pages 160–169. IEEE, 2011.

[24] N. Siegmund, M. Rosenmueller, M. Kuhlemann, C. Kaestner, S. Apel, and G. Saake. Spl conqueror: Toward optimization of non-functional properties in software product lines. *Software Quality Journal*, 20(3-4):487–517, 2012.

[25] N. Siegmund, M. Rosenmüller, C. Kästner, P. G. Giarrusso, S. Apel, and S. S. Kolesnikov. Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. *Information and Software Technology*, 55(3):491–507, 2013.

[26] J. Sincero, W. Schröder-Preikschat, and O. Spinczyk. Approaching non-functional properties of software product lines: Learning from products. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 147–155. IEEE, 2010.

[27] L. Sion and D. Van Landuyt. Systematic Quality Trade-off Support in the Software Product-Line Configuration Process: Supporting Site. https://people.cs.kuleuven.be/laurens.sion/splc2016/, March 2016.

[28] O. Sokolsky, I. Lee, and D. Clarke. Schedulability analysis of AADL models. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8–pp. IEEE, 2006.

[29] The International Organization for Standardization (ISO). Road vehicles – functional safety (iso 26262). http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=43464, November 2011. Part 1–10.

[30] T. Thuem, C. Kaestner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. Featureide: An extensible framework for feature-oriented software development. *Sci. Comput. Program.*, 79:70–85, Jan. 2014.

[31] D. Van Landuyt, S. Op de beeck, A. Hovsepyan, S. Michiels, W. Joosen, S. Meynckens, G. de Jong, O. Barais, and M. Acher. Towards managing variability in the safety design of an automotive hall effect sensor. In *Proceedings of the 18th International Software Product Line Conference*, September 2014.

[32] G. Zhang, H. Ye, and Y. Lin. Quality attribute modeling and quality aware product configuration in software product lines. *Software Quality Journal*, 22(3):365–401, 2014.

[33] H. Zhang, S. Jarzabek, and B. Yang. Quality prediction and assessment for product lines. In *Advanced Information Systems Engineering*, pages 681–695. Springer, 2003.